FACTFILE:
GCSE DIGITAL TECHNOLOGY
Unit 4
DIGITAL AUTHORING CONCEPTS

Rewarding Learning

Simple Error Handling Techniques

Learning Outcomes

Students should be able to demonstrate an understanding of and use the following:

- data validation, including presence, length, type and format checks;
- detection and correction techniques for syntax, execution and logic errors; and
- simple error trapping techniques.

Content

- Data validation
  - Presence Check
  - Type Check
  - Length Check
  - Format Check
- Programming Errors
  - Syntax errors
  - Execution Errors
  - Logic errors
- Fun with Debugging!
- Exception Handling

Data Validation

There is an old saying in computing: "*Garbage in garbage out.*" – GIGO. This refers to the fact that a computer system cannot produce correct results from incorrect data. This is – or ought to be – obvious to anyone who understands computers. It is, however, not always obvious to the general public.

It is not, in general, possible for a computer system to determine whether or not the data that it has been given is *correct*. It is, however, possible for it to determine, in some circumstances, whether or not the data is *reasonable*, given its intended meaning. This process can be automated and is known as *data validation*.

Data validation

"[Data validation] is the automatic checking of data entered into a computer system. Validation involves using the properties of the data to identify any inputs that are obviously wrong. Validation only proves that the data entered is a reasonable value for the computer to accept. It cannot prove that the data entered is the actual value the user intended."

BCS Glossary of Computing, 13th edition, p 75.

For example, if a user enters the value –5 when asked to enter their age, then this is an obvious error (age can't be negative), and an automated

check should be able to spot it. Similarly, 33 April 2017 is clearly incorrect if a date was expected – again an automated check should be able to spot this. We refer to these as *invalid values* and computer systems should have *validation checks* that prevent invalid values being entered.

Validation checking cannot of course catch all errors. For example, a user may mistakenly enter the value 21 as their age, when they are actually 22. Errors of this kind cannot generally be caught by validation checks.

The following kinds of validation check are commonly used: presence checks, length checks, type checks and format checks.

Presence Checks

In entering data, it may sometimes be acceptable to leave out some values. For example, in collecting a user's contact details it may be acceptable for the user to leave the telephone number field blank. On the other hand, other fields may be compulsory – for example, there is little value in collecting contact details without the user's name. A *presence check* is a validation check that ensures that data has been entered into all *compulsory fields*.

Length Checks

It is often possible to anticipate appropriate lengths for certain text values. For example, it might be reasonable to say that a user's name must contain more than 1 character and less than (say) 20. A *range check* is a validation check that ensures that data values are of appropriate *length*.

Type Checks

Data values are normally required to be of specific types. For example, a user's age must be numeric, while their name must be text. A *type check* is a validation check that ensures data values are of the appropriate *type*.

Format Checks

Sometimes data is required to be entered in a predefined pattern or format. For example, dates are often entered as three pairs of digits such as 27/05/17. A *format check* is a validation check that ensures data values are in the expected *format*.

Programme Errors

Errors in a computer program are sometimes known as *bugs*. Anyone who has developed more than the very simplest of computer programs will know that errors are inevitable and that finding and fixing bugs is an important part of the software development task.

> "If debugging is the process of removing software bugs, then programming must be the process of putting them in." ~ Edsger Dijkstra
>
> http://www.azquotes.com/quote/561997

Most computer program errors may be classified as either: syntax errors, execution errors or logic errors.

Syntax Errors

Syntax errors are a bit like spelling, punctuation or grammatical errors that you might make when writing in a natural language such as English.

Syntax Errors

"[Syntax errors] occur either when the program statements cannot be understood because they do not follow the rules laid down by the programming language, *statement syntax errors*, or when program structures are incorrectly formed, *program syntax errors* or *structure errors*."

BCS Glossary of Computing, 13th edition, p 301.

Execution Errors

While syntax errors can often be detected before or during compilation, some errors don't become obvious until the program is executed. These are called *execution errors* or *run-time errors*.

Execution Errors

"[Execution errors] or run-time errors are errors that are detected during program execution. These errors, such as overflow and division by zero, can occur if a mistake is made in the processing algorithm or asa result of external effects not catered for by the program, such as lack of memory or unusual data."

BCS Glossary of Computing, 13th edition, p 301.

Logic Errors

A *logic error* is an error in program design that is then carried forward into the code. A program with a logic error will generally compile and execute

without any indication that the error is present. It will not, however, produce the correct result.

Logic Errors

"[Logical errors] are mistakes in the design of a program, such as the use of an inappropriate mathematical formula or control structure, recognised by incorrect results or unexpected displays."

BCS Glossary of Computing, 13th edition, p 301.

Fun with Debugging!

The following short Python program is a simple calculator. It is intended to prompt the user for two numbers and an operation, apply the operation to the numbers and print the result. The operations are chosen from: addition (add), subtraction (sub), multiplication (mul) and division (div).

```
n1 = input('Type first number: ')
n2 = input('Type second number: ')
op = input('Type an operation [add, sub, mul or
div]: ')
if op = 'add':
   n = n1 + n2
elif op == 'sub':
   n = n1 – n2
elif op == 'mul':
   n = n1 * n2
else op == 'div':
  n = n1 / n2
print( N )
```

If you attempt to run this program in its current form Python will alert you to a syntax error in the condition on the first leg of the select statement (op='add'). Python can point out that there is an error but it cannot correct your code – this is the job of the programmer. In this case, we have used the assignment operator (=) when we should have used the comparison operator (==). This is corrected below.

```
n1 = input('Type first number: ')
n2 = input('Type second number: ')
op = input('Type an operation [add, sub, mul or
div]: ')
if op == 'add':
   n = n1 + n2
elif op == 'sub':
   n = n1 – n2
elif op == 'mul':
   n = n1 * n2
else op == 'div':
   n = n1 / n2
print( N )
```

If you attempt to run this version, Python will again alert you to a syntax error – this time in the final leg of the select statement (op == 'div'). The reason that this is an error is that this condition is introduced by the keyword *else*, but *else* should not be followed by a condition. If you want a condition on here you must use the keyword *elif*. This is corrected below.

```
n1 = input('Type first number: ')
n2 = input('Type second number: ')
op = input('Type an operation [add, sub, mul or
div]: ')
if op == 'add':
   n = n1 + n2
elif op == 'sub':
   n = n1 – n2
elif op == 'mul':
   n = n1 * n2
elif op == 'div':
   n = n1 / n2
print( N )
```

If you attempt to run this version of the program you will get a little further this time. Here is an example interaction.

```
Type first number: 2
Type second number: 3
Type an operation [add, sub, mul or div]: add
Traceback (most recent call last):
   File "d:\Users\Norman\Desktop\temp.py", line
15, in <module>
      print( N )
NameError: name 'N' is not defined
>>>
```

The prompts are displayed and you can type in some data but then an error occurs. Python's error report is shown in bold.

This is called a NameError: the program is attempting to print a variable (N) which has not been assigned a value. At this point the programmer/debugger should realise that it has used N (uppercase) when she should used n (lowercase). This is corrected below.

```
n1 = input('Type first number: ')
n2 = input('Type second number: ')
op = input('Type an operation [add, sub, mul or
div]: ')
if op == 'add':
```

```
    n = n1 + n2
elif op == 'sub':
  n = n1 – n2
elif op == 'mul':
    n = n1 * n2
elif op == 'div':
    n = n1 / n2
print( n )
```

This version of the program will now run without raising any errors. For example:

```
Type first number: 2
Type second number: 3
Type an operation [add, sub, mul or div]: add
23
>>>
```

Unfortunately there is still a problem with the code – the result should be 5 – not 23! An experienced programmer/debugger will realise the error results from the fact that the function, *input*, returns a string and not a numeric type. Consequently the '+' operator performs string concatenation rather than arithmetic addition. This is corrected below.

```
n1 = int(input('Type first number: '))
n2 = int(input('Type second number: '))
op = input('Type an operation [add, sub, mul or
div]: ')
if op == 'add':
    n = n1 + n2
elif op == 'sub':
    n = n1 – n2
elif op == 'mul':
    n = n1 * n2
elif op == 'div':
    n = n1 / n2
print( n )
```

Based on the following interaction you may be tempted to think that the program is now error free...

```
Type first number: 2
Type second number: 3
Type an operation [add, sub, mul or div]: add
5
>>>
```

...but you would be wrong! Effective testing must use a range of input data, and test all paths through a program. What happens if, for example, the user makes a typing error?

```
Type first number: 2
Type second number: 3
Type an operation [add, sub, mul or div]: dib
Traceback (most recent call last):
   File "d:\Users\Norman\Desktop\temp.py", line
15, in <module>
      print( n )
NameError: name 'n' is not defined
>>>
```

In this interaction the user typed 'dib' instead of 'div'. Since this does not match any of the legs of the select statement, control passes to the print statement without a value being assigned to n – hence the error. The code can be restructured to correct this error but a better way is to use Python's exception handling. This is described in the next section.

Exception Handling

A NameError in Python is an *execution error* – sometimes called an *exception*. When an error such as this occurs we say that an *exception has been raised*. An *exception handler* is a section of code that is designed to trap these errors.

The most straightforward way to deal with this exception is to use Python's *try-except* construct. This construct tells Python to *try* to complete a task but also specifies alternative code that is invoked if the event of an error. The program below includes an exception handler for the NameError.

```
n1 = int(input('Type first number: '))
n2 = int(input('Type second number: '))
op = input('Type an operation [add, sub, mul or
div]: ')
if op == 'add':
    n = n1 + n2
elif op == 'sub':
    n = n1 – n2
elif op == 'mul':
    n = n1 * n2
elif op == 'div':
    n = n1 / n2
try:
    print( n )
except NameError:
    print('Operation not recognised.')
```

The effect of the try-except statement is to print n in the normal way providing that there are no errors. If, however, there is an NameError, the string 'Operation not recognised.' is printed instead.

So – is the program now correct? Well, consider this interaction:

```
Type first number: 2
Type second number: 0
Type an operation [add, sub, mul or div]: div
Traceback (most recent call last):
  File "d:\Users\Norman\Desktop\temp.py", line 13, in <module>
    n = n1 / n2
ZeroDivisionError: division by zero
```

This time an exception is raised because the program is attempting to divide by zero. We can fix this by creating another exception handler.

```
n1 = int(input('Type first number: '))
n2 = int(input('Type second number: '))
op = input('Type an operation [add, sub, mul or div]: ')
try:
  if op == 'add':
    n = n1 + n2
  elif op == 'sub':
    n = n1 – n2
  elif op == 'mul':
    n = n1 * n2
  elif op == 'div':
    n = n1 / n2
  print( n )
except NameError:
  print('Operation not recognised.')
except ZeroDivisionError:
  print('Cannot divide by zero.')
```

Two changes have been made here. The obvious one is the creation of a new exception handler to deal with division by zero errors. The less obvious one is that the keyword, *try*, has been moved to an earlier point in the program. This is because the try-except construct only works for exceptions that are raised within the *try* section.

Question: Is the program correct now?

Question: How do you know?

Resources

- BBC Bitesize, Writing Error-Free Code: http://www.bbc.co.uk/education/guides/zcjfyrd/revision
- TutorialsPoint, Python 3 Tutorial: https://www.tutorialspoint.com/python3/index.htm
- TutorialsPoint, Python 3 – Exception Handling: https://www.tutorialspoint.com/python3/python_exceptions.htm